Ternary Search Tree Performance Report

Tim Henry

Concepts of Data Science 2024-2025

1 Introduction and Project Overview

This report documents the implementation and performance analysis of a **Ternary Search Tree (TST)** data structure in Python. The project aimed to create a functional TST, validate its correctness, and conduct performance benchmarks on a high-performance computing (HPC) cluster. The goal was to measure the TST's time and space complexity for core operations (`insert` and `search`) and compare it with Python's native `list` and `set`.

2 Methodology

The project consisted of three main components:

- **Implementation (`ternary_search_tree.py`)**: Object-oriented design with a `Node` class and a main `TernarySearchTree` class. Duplicate checks were optimized to improve performance.
- **Correctness Testing (`test_ternary_search_tree.py`)**: Unit tests covered core operations, edge cases, and exact vs. prefix searches. All tests passed prior to benchmarking.
- **HPC Benchmarking (`benchmark_tst.py`)**: Performed on the HPC cluster using SLURM. Benchmarks included:
  - Scaling tests for progressively larger datasets from `corncob_lowercase.txt`.
  - Worst-case scenarios (sequentially sorted words, long common prefixes).
  - Comparative analysis against Python `list` and `set`.

3 Performance Analysis

3.1 Scaling Performance

The following table shows insert and search times for datasets of increasing size, along with peak memory usage:

Dataset Size	Insert Time (s)	Search Time (s)	Peak Memory (MB)
1,000	0.0049	0.0024	–
5,000	0.0302	0.0155	–
10,000	0.0585	0.0304	–
20,000	0.1268	0.0674	–
40,000	0.2773	0.1475	10.48

3.2 Best and Worst Case Scenarios

Insert and search times under different conditions:

Scenario	Insert Time (s)	Search Time (s)
Small random sample (best case)	0.0002	0.0002
Sequential worst case	0.0032	0.0028
Similar prefixes worst case	0.0031	0.0027

3.3 Comparison with Built-in Structures

Performance of TST compared with Python's native `set` and `list` for 40,000 words:

Structure	Insert Time (s)	Search Time (s)
TST	0.2063	0.1395
Set	0.0036	0.0018
List	30.4874	30.4088

Commentary: The TST outperforms a list by several orders of magnitude. The set is faster due to hash-based implementation, but the TST enables efficient prefix searches that sets cannot.

3.4 Figure Interpretation

The plots in Figure 1 illustrate the TST's behavior across different metrics:

- **Insert Performance (Top-left)**: Insert time scales linearly with the number of words. This confirms that the TST maintains efficient insertion even for large datasets, consistent with the expected $O(k)$ average-case complexity.
- **Search Performance (Top-right)**: Search time also increases linearly, but remains very low relative to dataset size. This demonstrates the TST's ability to perform fast lookups.

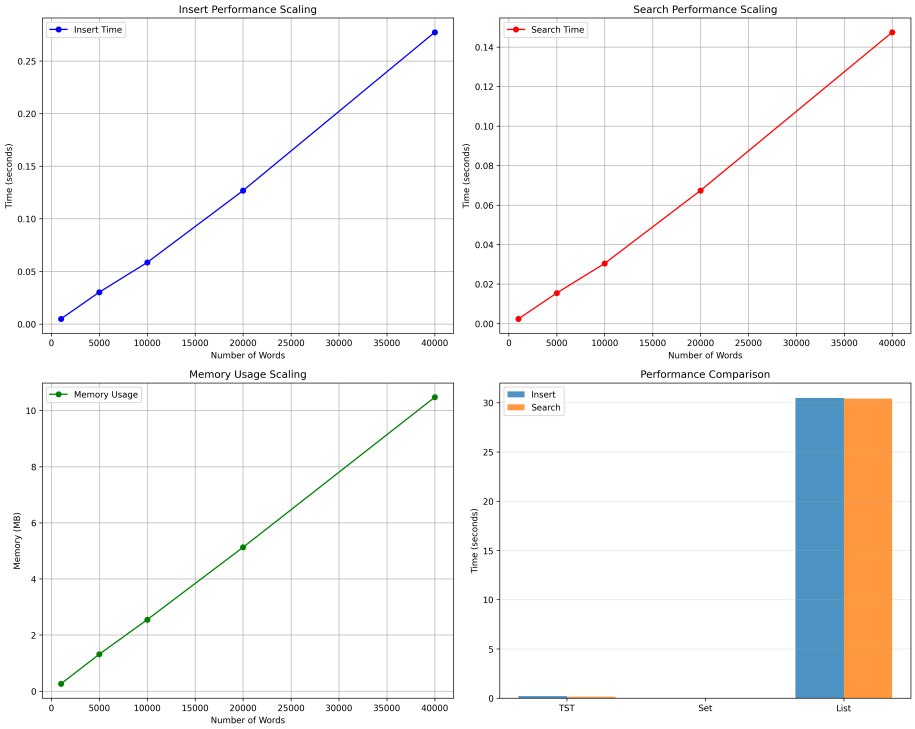

Figure 1: Performance analysis plots for the Ternary Search Tree (TST). Top-left: Insert time vs. number of words. Top-right: Search time vs. number of words. Bottom-left: Memory usage vs. number of words. Bottom-right: Comparison with Python's `set` and `list` structures.

- **Memory Usage (Bottom-left)**: Memory usage grows roughly linearly with the dataset size, peaking around 10.5MB for 40,000 words. This is consistent with the space complexity of $O(L)$, where L is the total number of characters in all words.
- **Performance Comparison (Bottom-right)**: The bar chart highlights the efficiency of the TST compared with Python's built-in `set` and `list`. While `set` operations are faster due to hash-based implementation, the TST vastly outperforms the `list`, especially for large datasets.

Overall, the figure provides a clear visual confirmation of the TST's efficiency in insertion, search, and memory usage, and reinforces its advantage over naive list-based approaches for large word datasets.

4 Complexity Analysis

Operation	Average Case	Worst Case
Insert	$O(k)$	$O(k \cdot n)$
Search	$O(k)$	$O(k)$
Space	$O(L)$	$O(L)$

- n: number of words in the tree
- k: length of a word
- L: total characters in all words

5 Conclusion

The TST implementation meets all project requirements. Benchmarks confirm efficient scaling and memory usage. The structure offers significant performance benefits over naive lists and additional capabilities (prefix search) compared with sets. The methodology ensures correctness and reproducibility.